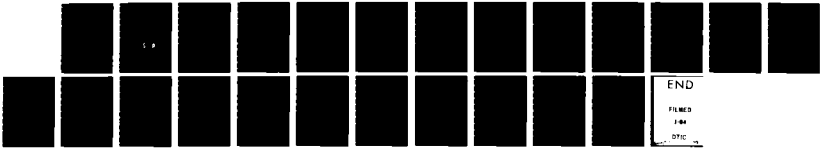
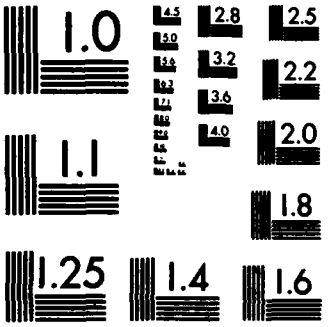
AD-A136 093 ELEMENTS OF KNOWLEDGE-BASED EXPERT SYSTEMS(U) DELAWARE 1/1
UNIV NEWARK DEPT OF COMPUTER AND INFORMATION SCIENCES
D CHESTER MAR 82 AFOSR-TR-83-1143 AFOSR-80-0190
UNCLASSIFIED F/G 6/4 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ELEMENTS OF KNOWLEDGE-BASED EXPERT SYSTEMS*

by

Daniel Chester

Department of Computer and Information Sciences
University of Delaware
Newark, DE 19711

March, 1982

*Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant no. AFOSR-80-0190. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copywright notation herein.

A reprint of a paper from Proceedings Micro-Delcon '82, University of Delaware, Newark, DE, March 9, 1982.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-83-1143	2. GOVT ACCESSION NO.: AD-A136 093
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): ELEMENTS OF KNOWLEDGE-BASED EXPERT SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED: ~~Interim technical report~~
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Daniel Chester	8. CONTRACT OR GRANT NUMBER(s): Grant #: AFOSR-80-0190
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer & Information Sciences, University of Delaware, Newark, DE 19711	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: March 1982
	13. NUMBER OF PAGES: 23
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

A reprint of a paper from Proceedings Micro-Delcon '82, University of Delaware, Newark, DE, March 9, 1982.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Expert systems, algorithms, control strategy, expert knowledge, natural language requirements, forward chaining, backward chaining, rules, agendas, indexing, frames, demons

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Expert systems are built to solve problems in application areas for which 'good' algorithms are not known. These systems consist of a global data base of assertions, a set of rules that represent small bits of an expert's knowledge, and a control strategy for applying the rules to the assertions. Agendas are used to make the control strategy more efficient. Efficiency can be further increased by indexing the rules and assertions in various ways, one of which is frames. A system for deriving formal specifications from natural language requirements is presented as an example.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Introduction

In recent years researchers in artificial intelligence have implemented many systems that they call 'expert systems'; these are systems designed to solve problems in the same ways that human experts do. Four expert systems have come into regular use: Dendral [1], Macsyma [2], Puff [3], and R1 [4]. (Dendral determines the molecular structure of organic compounds; Macsyma manipulates algebraic expressions symbolically, including their integration and differentiation; Puff diagnoses pulmonary disorders; and R1 configures VAX systems.) This paper discusses i) the kind of problem expert systems try to solve, ii) the representation of expert knowledge, iii) the application of that knowledge, and iv) the organization of that knowledge for efficient operation. The examples in this paper are based on an expert system for deriving formal specifications from natural language requirements. Drs. Weischedel and Chester are developing this system at the University of Delaware.

Application areas

A 'good' algorithmic solution to a problem is a step-by-step procedure that is based on a well-understood theory of the problem domain. Such a solution usually works in polynomial time; that is, the time taken to solve a problem is proportional to a polynomial function of the size of the problem data. It is a method frequently used by people who must solve the problem every day. For instance, the Simplex method is commonly used to solve

linear programming problems; various numerical integration methods are often used to solve differential equations; discrete event simulation techniques are often used to obtain certain information about complex processes. Because a 'good' algorithmic solution to a problem is based on a well-understood theory, it can be taught; anyone can become an 'expert' at solving this type of problem.

Expert systems solve problems that don't have 'good' algorithmic solutions, or whose algorithmic solutions are inefficient or difficult to follow. For example, we don't know how doctors diagnose medical conditions; Mycin [5] is an expert system that does this for blood diseases. Similarly, no 'good' algorithmic solution is known to the problem of determining the molecular structure of organic compounds from their chemical formulas, together with mass spectrograph and nuclear resonance data, which is what Dendral [1] does. Likewise, there are no 'good' solutions known to the problems of understanding continuous speech, discovering new mathematics, or designing electronic circuits. But expert systems have been built to solve all these problems [6,7,8], at least to some extent.

People become expert at many kinds of problems without having a well-understood theory, but to do so, they train for a long time. Doctors study medical facts for many years and acquire much knowledge from experience before they are experts at diagnosis. Engineers also study many facts about their specialties and 'learn by doing'. The same holds for programmers. Doctors, engineers and programmers don't have well-understood theories for their problem domains, but they do solve their problems

methodically. They devise their own techniques and shortcuts for effectively applying the facts they learned in school and from experience. Expert systems attempt to implement techniques and shortcuts similar to these.

Most expert systems consist of three parts: a global data base in which is stored the data that describes the problem to be solved and all the intermediate results in its solution, a set of rules that encode the knowledge of a human expert, and a control strategy that applies the rules repeatedly to the data until the problem is solved. Rules may be applied mainly in the forward or in the backward direction; some systems apply them in both directions. Only the strictly backward chaining and the strictly forward chaining strategies will be discussed here.

Global data base

The global data base holds the problem-specific data in the form of assertions. These are expressions in a formal language; often they are atomic expressions consisting of a predicate name and a list of arguments. These assertions represent information in a way that is independent of where they are located in the computer. They are thus different from the data that is stored in an array, where the significance of an array element is a function of the array and the index value by which it is accessed.

As an example of the assertions that might be in the global data base, suppose that the system for deriving formal specifications mentioned above were given the sentence "An ordered list is

empty" as part of its input. Let this sentence and its components have the following names:

S: "an ordered list is empty"
P: "is empty"
N: "an ordered list"
R: "ordered"
H: "list".

The parser produces a representation that gets stored in the global data base as the set of (paraphrased) assertions

```
P is the grammatical predicate of S
the translation of P is the formula 'empty(X)'
N is the grammatical subject of S
R is the restriction on the head word of N
the translation of R is the formula 'ordered(X)'
H is the head word of N
the translation of H is the formula 'list(X)'
'X' is the variable in the translation of N.
```

(To simplify the illustration, we take the sentence to mean "every ordered list is (always) empty" and show only a subset of the actual assertions.)

We retrieve the assertions in the global data base by giving a pattern against which they are matched. These patterns are partial descriptions; they include variables where the components of the desired assertions are unknown. The accessing function retrieves the assertions that satisfy these descriptions one at a time and assign the corresponding components to the variables. (This matching process is frequently some variation of the unification algorithm [9].) If we query the above set of assertions with the pattern

the translation of P is the formula Z,

we obtain the assertion

the translation of P is the formula 'empty(X)'.

The variable Z now has the value 'empty(X)'. If we ask instead

the translation of Y is the formula Z,

we obtain the same assertion, giving Z the same value as before and giving variable Y the value P. If we ask the same question again, we obtain

the translation of R is the formula 'ordered(X)'

giving Y the value R and Z the value 'ordered(X)'. On asking a third time, we obtain

the translation of H is the formula 'list(X)'

giving Y the value H and Z the value 'list(X)'. A fourth asking of the question results in failure of the accessing function.

The other major operations on the data base are insertions and deletions. All of these operations manipulate the assertions as if they were stored in a featureless bag and pulled out by magic, except that the order in which assertions matching a pattern are pulled out depends on the order in which they were stored.

Rules

The knowledge that an expert brings to a problem is represented in an expert system by a set of rules. These rules, which are frequently obtained from cooperating experts, take the form of a conditional statement:

IF premiss THEN conclusion

where both the premiss and the conclusion may be simple expressions or combinations (usually conjunctions) of simple expressions. To illustrate, here are two rules for translating simple sentences into formal logic expressions. The terms Xs, Xp, Xpt, Xn, Xnt, and V in these rules are variables; they will be assigned values when the rules are applied; in particular, Xs will be assigned the name of a sentence.

```
IF Xp is the grammatical predicate of Xs &
   the translation of Xp is the formula Xpt &
   Xn is the grammatical subject of Xs &
   the translation of Xn is the formula Xnt &
   V is the variable in the translation of Xn
THEN the translation of Xs is the formula
      '(V)(Xnt -> Xpt)'

IF Xn is the grammatical subject of Xs &
   Xh is the head word of Xn &
   the translation of Xh is the formula Xht &
   Xr is the restriction on the head word of Xn &
   the translation of Xr is the formula Xrt
THEN the translation of Xn is the formula
      '(Xht & Xrt)'
```

(When the rules are applied, the variables in the quoted expressions are replaced by their values to produce the intended formula.)

The rules in an expert system represent the individual bits of knowledge accumulated by an expert. Each one is a tiny generalization, a single insight that the expert has learned. Human experts are seldom conscious of these rules until they carefully introspect about what they do to solve problems. These bits of knowledge are acquired over a period of years with little attempt to organize them into step-by-step procedures. Consequently, these bits of knowledge (and the rules that represent them) contain no explicit "flow-of-control" to guide their application, which is quite different from modern computer programs! To compensate for this lack, expert systems include control strategies for applying the rules; in these systems flow of control information is completely divorced from application area information.

A rule can be applied in basically two ways. If an expert knows that the premiss is true, he can infer that the conclusion is true. This is known as forward chaining. Alternatively, if he wants to find out if some statement is true and the conclusion of the rule implies that desired statement, he can try to establish that the premiss is true. This is known as backward chaining. It may turn out that many rules can be applied, so the expert has to decide which to go ahead with. Another property of many of these rules is that they are not certain; the conclusion is only a likely consequent of the premiss.

Backward chaining

Some expert systems backward chain their rules; they reduce goals that match conclusions of rules to the subgoals stated in

the corresponding premisses. This approach resembles conventional programming in that it operates in a top-down, goal-directed way that is similar to procedure calls. All the rules that have the same predicate name in their conclusions can be thought of as defining a procedure by that name. This procedure is defined as a single nested IF-THEN-ELSE statement in which each IF-THEN branch implements one rule. So if three rules define a procedure, the body of that procedure would have the form

```
     IF premiss1 THEN RETURN conclusion1
ELSE IF premiss2 THEN RETURN conclusion2
ELSE IF premiss3 THEN RETURN conclusion3.
```

These expert systems also include a backtracking facility, which means that once a branch is taken, the procedure is exitted, but if a failure occurs, the procedure is reentered and another branch is taken if possible. Prolog [10] is a programming language that has this backtracking facility built into it.

To see how backward chaining of rules works, consider the rules and assertions shown above. To find out what the translation of sentence S is, the system is given the goal of obtaining an instance of the assertion

the translation of S is the formula Z.

This matches the conclusion of the first rule by giving Xs the value S and Z the value '(V)(Xnt -> Xpt)'. Each of the simple assertions in the premiss is then obtained. The first three are obtained by matching them against the assertions in the data base, giving Xp the value P, Xpt the value 'empty(X)', and Xn the

value N. The fourth simple expression does not match an assertion; in this case, the bound variable Xn is replaced by its value and the resulting expression is matched against the conclusion of the second rule. The expressions in the premiss of this rule match assertions in the data base, giving Xnt the value '(list(X) & ordered(X))'. Finally, the last simple expression in the premiss of the first rule matches an assertion, giving V the value 'X'. By replacing all variables by their values, the goal pattern becomes

```
the translation of S is the formula
'(X)((list(X) & ordered(X)) -> empty(X))'.
```

The example rules and assertions don't show how backtracking works, but if the value obtained for Z were unsatisfactory and there were other translation rules, the system would throw away the variable assignments, apply the other rules to obtain a new value for Xnt, and proceed again as before.

Forward chaining

Other expert systems forward chain their rules. The rules are like condition-action statements; when the condition described in the premiss of a rule holds, the conclusion describes a possible action that might be taken. The conclusion is likely to contain several expressions, each of which gives a simple operation on the data base: insert or delete an assertion, print to or read from the terminal, etc.. Since a particular set of assertions in the data base may satisfy the conditions of many rules, these expert systems have strategies for selecting the rule whose

actions will actually be carried out. Nearly all follow the principle that once a set of assertions activate a rule, that rule is not activated again by that set until at least one of the assertions gets temporarily removed from the data base.

For our example rules, forward chaining is particularly simple. As the example assertions are put into the global data base, they are matched against the expressions in the premisses of the rules in every way possible. When the last fact is added, a combination of matches will be found that satisfies the entire premiss of the second rule, causing the conclusion

```
the translation of N is the formula
     '(list(X) & ordered(X))'
```

to be added to the data base. When so added, it too gets matched against the expressions in the premisses, satisfying the premiss of the first rule, which in turn causes the conclusion

```
the translation of S is the formula
'(X)((list(X) & ordered(X)) -> empty(X))'
```

to be added. Since these are the only conclusions that can be reached through these rules, the system stops computing and waits for the input of more assertions.

Forward chaining systems have the advantage that they can respond quickly to sudden changes in their environment; they won't keep on working on the same problem when sudden changes make the problem no longer important or a new problem more important. Such readiness to change the focus of attention is important in real-time systems.

In some ways building a forward chaining system is like programming with decision tables. Only here there are many more

conditions to test and not every combination of conditions has its outcome specified in a rule.

Agendas

A simple conceptual model of a forward chaining system is that it operates by repetitively executing a Recognize-Resolve-Act cycle. In the Recognize phase of each cycle, the conditions of all the rules are evaluated to see if they are satisfied by the current assertions in the global data base. Rules with satisfied conditions indicate competing, possibly conflicting actions that can be taken at this time. In the Resolve phase, one of these rules is selected. Often, in simple systems, this is just the first rule found with a satisfied condition. In more sophisticated systems each rule with a satisfied condition is given a priority; this priority is a function of such things as when the rule was added to the system, when the assertions that satisfy the condition were added, and the a priori importance of the actions of the rule to the overall purpose of the system. In the Act phase, the actions of the selected rule are carried out. This cycle is repeated over and over until a terminating action is taken or no more rules get activated.

This conceptual model is a convenient way to imagine how a forward chaining expert system operates, but it is quite inefficient if implemented directly. The most redundant part of the model's operation is the repetitive evaluation of the conditions in order to determine which rules to activate. This part of the operation can be made more efficient by putting the rules with

satisfied conditions on a list, or agenda, where they remain until chosen for activation or until their conditions cease to be satisfied, due to a change in the global data base. If all of the satisfied rules are on the agenda and a new assertion is added to (or deleted from) the data base, only the rules which require the presence (or absence) of that assertion need to have their conditions evaluated.

The behavior of such an expert system depends critically on the manner in which it puts potentially activated rules on the agenda. In some systems, such as AMORD [11], the agenda is a queue; the rules are put on the back of the agenda and the rule at the front is selected for activation. This approach is useful when the expert system infers consequences of its initial assertions; it mostly adds new assertions to the data base and only rarely deletes them. Under these circumstances, every rule that can be activated should be activated eventually. A queue guarantees this; a stack would be unsatisfactory, since if the agenda were a stack, it would be possible for some rules to dominate the system, causing themselves to be repeatedly added to the stack so that it eventually overflows or at least goes into an infinite loop of activity.

Some systems, like Hearsay II [6] and AM [7], compute a priority for each rule before it is inserted into the agenda. In these systems, each rule has explicit data or code for computing this priority. The priority of a rule may change with time as the global data base changes so that the rule is moved in the agenda, which is always kept in priority order.

A few systems, such as OPS4 [12] and OPS5 [13], assign a

priority to rules implicitly; the rules do not contain any data (such as ratings) expressly for computing priority. These systems select the rule with the most recently satisfied and most specific condition. This approach relieves the implementer of the task of supplying arbitrary functions for computing rule priority.

Indexing

The process of recognizing the rules to be activated can be speeded up by suitably indexing the assertions in the global data base and the rules in the system. Simple assertions that consist of a predicate name and some arguments can be grouped and indexed by the predicate name or by one of the arguments. If the predicate name is the indexing term (such as in Prolog) the group of assertions having that predicate name is like a table in a relational data base, with each assertion corresponding to one line in the table. This mode of indexing is useful when collections of similar facts are desired, such as finding out which employees will retire in ten years. In expert systems, however, what is desired is often complex information about individual persons and things. In these systems the assertions are indexed on one or more of the arguments, which are names of objects. When the assertions are limited to predicate names having exactly two arguments and are indexed on both arguments, the global data base is often called a semantic network, after the terminology of some early work [14] in which this organization was used to represent the definitions of words.

A closely related way to organize the global data base is to make it be a collection of entities called frames. Each frame is a group of assertions about an object, only now each assertion is a 4-tuple of the form

object-name relation aspect 2nd-argument

where the relation is often called a slot and the aspect is often called a facet [15]. In a frame-based system our example assertions about the head word H of grammatical subject N might be expanded into the 4-tuples

```
H    head         VALUE     N
H    translation  VALUE     'list(X)'
H    translation  MUST-BE   formula
H    translation  DEFAULT   'thing(X)'.
```

These 4-tuples assert, respectively, that H is the head (word) of N, the VALUE of the translation of H is 'list(X)', the translation of H MUST-BE a formula, and if no VALUE of the translation of H is stored, the DEFAULT VALUE is the formula 'thing(X)', which just says that whatever is being talked about is a thing. For efficiency the assertions about one object are combined into a single tree structure indexed under the object name. For the example above, the tree structure would look like this:

```
H\-----head-------------VALUE-------N
  \-----translation\-----VALUE-------'list(X)'
                    \-----MUST-BE-----formula
                     \-----DEFAULT-----'thing(X)'.
```

The rules in an expert system are organized in various ways. One of the simplest is to select one of the simple expressions in the premiss of a rule to be its trigger; the process that adds an

assertion to the data base also checks to see if it matches any triggers. Whenever a trigger matches, the premiss of the corresponding rule is tested; if the test succeeds, the rule is put on the agenda, or, in systems without an agenda, is activated immediately. If the premiss of a rule contains several simple expressions, every one that might be added last to the data base must be made a trigger in a copy of the rule. (Each copy can have its trigger assertion removed from the premiss to avoid duplicate testing.) If the third simple expression in the second example rule were chosen to be a trigger, the trigger-rule pair would be

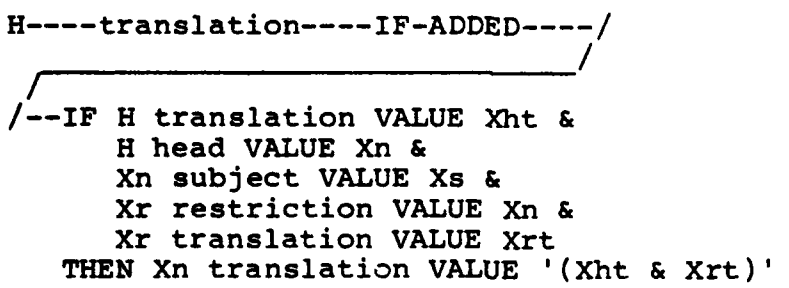
```
TRIGGER:
 the translation of Xh is the formula Xht

RULE_COPY:
 IF Xh is the head word of Xn &
    Xn is the grammatical subject of Xs &
    Xr is the restriction on the head word of Xn &,
    the translation of Xr is Xrt
 THEN the translation of Xn is the formula
           '(Xht & Xrt)'.
```

Since some rules might test for the absence of assertions, the process that deletes assertions from the data base also checks a set of triggers that are assertions that must be absent; when a trigger matches, the premiss of the corresponding rule is tested and the rule put on the agenda or activated accordingly, just as in the additive case.

In older knowledge-based systems [16] these trigger-rule pairs were called demons because they were like demons that would constantly watch the contents of the data base, waiting for the conditions to be just right so they could do their mischief!

In frame-based systems demons are stored directly in the

frames. A trigger pattern is an assertion about the VALUE aspect of a relation; if the assertion must be present, the rule is stored under the IF-ADDED aspect of the relation, and if it must be absent, the rule is stored under the IF-REMOVED aspect. For instance, the trigger-rule pair above could be instantiated on H (H is substituted for Xh) and stored in the frame for H; this would add the path



to the tree shown above. (The expressions in the rule are written as 4-tuples so that they can match the assertions in a frame-based system. The trigger pattern is included in the premiss in order to assign a value to variable Xht.)

To avoid having a separate copy of a rule for each object of the same type, the rule is stored in the frame representing the type and each object is related to the type by an AN-INSTANCE-OF relation or an A-KIND-OF relation. The processes that add and delete assertions check the type-frames of an object as well as the object-frame in their search for demons to activate. Frame-based systems can also hold backward chaining rules, which are stored under the IF-NEEDED aspect of the relations that the rules are meant to compute.

Example system

The assertions and rules shown in this paper are based on an expert system for deriving formal specifications from natural language requirements. This system consists of three components: a natural language parser, a discourse analyzer and a concept mapper. The natural language parser used is the RUS parser [17]; it backward chains an extensive set of rules for English syntax (that we augmented with more conditions and actions) to obtain a sentence parse and semantic representation. For the sentence "An ordered list is empty", the RUS parser produces the list structure

```
[DECLARATIVE
  AUX =
    [(PRESENT)]
  FRAMETYPE = <PROPOSITION>
  TRANSL =(EMPTY X)
  SUBJECT =
    [NP
      DET =
        (ART...AN...)
      FRAMETYPE = LIST
      RESTRICTIONS =
        [(ORDERED X)]
      TRANSL = (LIST X)
      VAR = X
      HEAD =(NOUN...LIST...)]
  HEAD =[ADJVERB
          HEAD = EMPTY]
  COPULA =(BE)]
```

which becomes the input to the discourse analyzer.

The discourse analyzer breaks this list structure down into the component assertions already shown and stores them in the global data base. At this point the individual words of the sentence have translations (produced by the parser). The rules in the discourse analyzer combine these into a translation for the

whole sentence. This process requires that terms referring to the same object be identified. When several sentences are processed at once, these terms can come from different sentences. The result of the discourse analyzer is a logic formula that represents the literal content of the sentences. For the example sentence, we get the formula

```
(X)((list(X) & ordered(X)) -> empty(X)).
```

Though the output of the discourse analyzer is a logic formula it may not be a suitable formal specification. If the input text mentions geometric metaphors like "left end of the list" or "add to the top of the stack", the discourse analyzer translates these metaphors literally. The concept mapper recognizes these metaphors and maps them into appropriate mathematical expressions. For example, if the input text defines "ordered list" in terms of finite sequences, the mapper maps the expression 'empty(X)' into 'X=nil', where nil is the empty sequence. If the definition were in terms of an array, the mapping of the expression would be different. The output of the concept mapper is a second logic formula with metaphors removed and gaps filled in, making it a suitable formal specification. (The gaps arise from incompleteness in the natural language requirement and are filled by rules in the mapper or by interaction with the user.)

Both the discourse analyzer and the concept mapper forward chain their rules. This approach was chosen because the processes are data driven rather than directed top-down. We only know what to do for specific cases; as we look at more examples we discover new cases and figure out what to do for them.

Forward chaining is appropriate for this situation because it allows the analyzer and mapper to do what they can with the data they receive. Without a global view, we cannot give these components a goal to work toward, which would be required by a backward chaining approach. Contrast this with the parser component, where an extensive English grammar does provide a global view of the parsing process and a backward chaining approach is practical.

Conclusion

Expert systems provide a way to approach applications that cannot be programmed by looking up some algorithms in a textbook. They are particularly useful where new knowledge has to be discovered and codified before program development is possible. The approach taken by these systems is to completely separate knowledge about the application area from the information about how to proceed. The application area knowledge is entered in the form of rules that resemble simple conditional statements. Each rule represents an independent piece of knowledge and is thus a kind of module that can be added or removed without much concern for its effect on the contents of other rules.

The control strategies in expert systems apply rules in two ways: backwards and forwards. Backward chaining resembles the recursive procedure calls of conventional programming, but it usually includes a backtracking feature that allows procedures to be re-entered and computed differently. Forward chaining allows expert systems to react quickly to changes in their data bases

and to operate without a definite goal, which might focus their 'attention' on too narrow a portion of their data. They can be more 'opportunistic' this way, taking advantage of any significant features of a particular problem.

If the rules were applied in the simplest way possible, expert systems would be quite inefficient. There are several techniques to increase their efficiency. These techniques consist mostly of ways of indexing rules and assertions for quick retrieval, scheduling rule applications for proper sequencing, and storing intermediate data structures to avoid repeated computations. Expert systems have become practical with the adoption of these techniques.

Acknowledgements

Toni Cohen, Tom Myers, Don Perlis and Ralph Weischedel made many valuable suggestions for improving the initial draft of this paper.

References

1. Lindsay, R. K., B. G. Buchanan, E. A. Feigenbaum and J. Lederberg. Applications of Artificial Intelligence for Organic Chemistry - The DENDRAL Project. McGraw-Hill, 1980.

2. Martin, W. A. and R. J. Fateman. The MACSYMA system. Proc. ACM 2d Symposium on Symbolic and Algebraic Manipulation, Los Angeles, CA, 1971, pp. 23-25.

3. Heuristic Programming Project 1980. Heuristic Programming Project, Computer Science Department, Stanford University, Stanford, CA.

4. McDermott, J. R1: an expert in the computer systems domain. Proc. of the First Annual National Conference on Artificial Intelligence, Stanford University, August 18 to 21, 1980, pp. 269-271.

5. Shortliffe, E. H. Computer-Based Medical Consultations: MYCIN. American Elsevier, New York, 1976.

6. Lesser, V. R. and L. D. Erman. A retrospective view of the Hearsay-II architecture. 5th International Joint Conference on Artificial Intelligence, 1977, pp. 790-800.

7. Davis, R. and D. B. Lenat. Knowledge-Based Systems in Artificial Intelligence. McGraw-Hill, New York, 1982.

8. Stallman, R. M. and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence, Vol. 9, No. 2, October 1977, pp. 135-196.

9. Charniak, E., C. K. Riesbeck and D. V. McDermott. Artificial Intelligence Programming. Lawrence Erlbaum Associates, Hilldale, New Jersey, 1980.

10. Warren, D. H. D. and L. M. Pereira. PROLOG: the language and its implementation compared with LISP. SIGPLAN Notices Vol. 12, No. 8 / SIGART Newsletter No. 64, August 1977, pp. 109-115.

11. de Kleer, J., J. Doyle, G. L. Steele, Jr. and G. J. Sussman. AMORD: Explicit control of reasoning. SIGPLAN Notices Vol. 12, No. 8 / SIGART Newsletter No. 64, August 1977, pp. 116-125.

12. Forgy, C. L. OPS4 User's Manual. Department of Computer Science, Carnegie-Mellon University, July 1979.

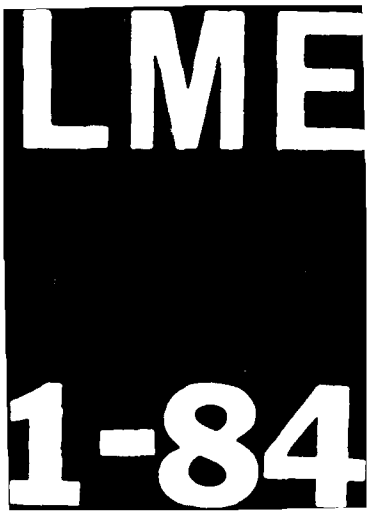
13. Forgy, C. L. OPS5 User's Manual. Department of Computer Science, Carnegie-Mellon University, July 1981.

14. Quillian, M. R. Semantic Memory. In Semantic Information Processing, M. Minsky, ed., MIT Press, Cambridge, MA, 1968, pp. 216-270.

15. Winston, P. H. and B. K. P. Horn. LISP. Addison-Wesley, Reading, MA, 1981.

16. McDermott, D. V. and G. J. Sussman. The CONNIVER Reference Manual. AI-M-259A, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1974.

17. Bobrow, R. J. The RUS system. In Research in Natural Language Understanding, B. L. Webber and R. Bobrow, eds., BBN Report No. 3878, Bolt Beranek and Newman Inc., Cambridge, MA, 1978.


LME
1-84